**Decorator**

Декоратор е структурен шаблон за дизайн, който се използва в обектно-ориентираното програмиране.Този шаблон може да бъде използван за разширяването на функционалността на опеределен клас по времето на изпълнение на програмата, като запазва интерфейса му. Това разширение може да бъде постигнато чрез прибавянето на нов клас които "обвива" оригиналния клас. Декораторите предоставят гъвкава алтернатива на наследяването за разширяване на функционалността.

Най-просто казано, идеята е – когато използваме определена йерархия от класове и искаме да добавим нова функционалност към няколко от тези класове, се налага за всеки наш клас да добавим по още един, или с други думи – класовете ни стават два пъти повече. Ако след това искаме да добавим още някаква функционалност върху всички съще съществуващи вече класове, класовете ни стават четири пъти повече. Решението е да използваме Декоратор, като направим един клас Decorator и от там нататък за всяка нова функционалност създаваме само по един нов клас.

Този нов клас пази като поле инстанция на интерфейса или абстрактния клас, който се наследява от нашата йерархия, а освен това сам по себе си този нов клас имплементира този интерфейс или абстрактен клас. Така чрез подходящо използване на конструкторите за подаване на параметри, можем да създаваме модифицирани обекти от всички наши класове в йерархията, без да увеличаваме експоненциално броя на класовете, които имаме. Друго голямо предимство е, че не се налага да променяме използването на старата функционалност – тя продължава да си работи по начина, по който е работила досега.

В примерната ми имплементация имам класове за автомобил и мотор. Обаче ако искам да добавя и спортен автомобил и спортен мотор, трябва да добавя класове и за тях. Ако имам бъги, ще ми трябва и спортно бъги и т.н. А ако всяко спортно превозно средство има две разновидности – офроуд и пистово, трябва да направя още толкова класове и да променя старата функционалност. Затова в случая се налага използване на декоратор.

**Observer**

Observer е поведенчески шаблон за дизайн, който се използва в обектно-0ориентираното програмиране. Използва се в случаите, при които група от обекти (наблюдатели), след регистрацията им към регистриращ обект биват оповестявани за промени в неговото състояние.

Най-очевидният пример от реалността са абонаментните услуги. Например даден вестник си има списък от абонати. При всяко издаване на вестника, списъкът от абонати се обхожда и те се информират, че вестника е издаден (като в случая получават вестника). Друг класически пример с конкретна имплементация на C# е този с пътници, които трябва да бъдат информирани за багажа им: <http://msdn.microsoft.com/en-us/library/ee850490(v=vs.110).aspx> .

В моята имплементация, която съчетава observer със singleton, имам списък от наблюдатели, които искат да бъдат информирани за промени в журнала за наличностите. Журналът за наличности ги абонира за себе си (добавя ги в собствен списък) и при всяка промяна в него, минава по всеки то абонатите и ги информира (в случая всеки абонат си записва промяната в свой списък).

**Singleton**

Singleton е създаващ шаблон за дизайн, който се използва в обектно-ориентираното програмиране. Този шаблон се използва обикновено в моделирането на обекти, които трябва да бъдат глобално достъпни за обектите на приложението (например обекта съдържайки структурите с настройките на програмата ) или обекти, които се нуждаят от максимално късна инициализация за пестенето на ресурси от паметта.

Singleton се използва тогава, когато искаме да си гарантираме, че технологично не е възможно да имаме повече от една инстанция от определен клас в нашето приложение. Идеята е – задаваме private конструктор на класа, с което си гарантираме, че няма публичен празен конструктор, който да може да се достъпва отвън. Самият клас пази частна статично поле от собствения си тип. Конструктора на класа е скрит, но статичното поле само по себе си е част от класа, затова достъпва конструктора и може да се инициализира. Важно е да се отбележи, че ако се пропусне static в имплементацията, ще се инициализира безкрайна рекурсия, тъй като всяко извикване ще извиква ново поле, което ще извиква ново поле, което ще извиква ново поле и т.н.. Това е често срещана грешка при имплементацията. Публичният интерфейс на singleton класа е метод, който връща статичната инстанция. Така се гарантира, че тя е една единствена, класът не може да се инстанцира отвън.

Разновидност на singleton е threadsafe singleton, който намира приложение при многонишкови приложения. В този случай си гарантираме, че ако приложението ни е многонишково, технологично е невъзможен сценарий, при който повече от една нишка да направи инстанция на класа и така да имаме повече от една инстанция. Постига се чрез двойно заключване на ресурса за останалите нишки.

В примерната ми имплементация имам журнал за наличности, в който може да се записват отделни обекти с дата, час, съобщение и движение – дали нещо е добавено или излязло от наличността. Тъй като журналът е един единствен за цялото приложение, възниква необходимостта той да бъде singleton.